Python Tutorial

Throughout this course, you will be using the Python programming language to implement various algorithms for your assignments. This tutorial is intended to give you a working knowledge of Python and cover some basic functionality that will be useful for implementing the algorithms discussed in this course.

This tutorial consists of 5 sections. You must go through these sections and complete any exercises. If there is something to turn in, the section will have submission instructions. Some sections simply go over some aspect of Python and there will be nothing to turn in. However, the best way to learn is by doing, so you should take the time to practice and explore the concepts that are discussed.

Alternatively, if you already have a workable knowledge of Python then you may skip the tutorial and present us the evidence that you have that knowledge.

Ultimately, you should submit a zipped folder called `python_tutorial` that contains (1) the `google-python-exercises` folder (from Section 0) and (2) `function_passing.py` (from Section 2).

Section 0: Google's Python Class (Broad Overview)

Estimated time: 8 hours
Reference Site: https://developers.google.com/edu/python/

Google has put together a course that presents the basics of python with a series of short lectures, written materials, and practice exercises. Go through all the written materials in parallel with the corresponding lecture videos and complete the exercises at the end of each section.

Submission Instructions:

- Add the entire `google-python-exercises` folder to your submission folder.

Section 1: Object Oriented Programming

Estimated time: 1 hour
Reference Site: https://docs.python.org/2/tutorial/classes.html

Python is a multi-paradigm language, meaning you can write code in a procedural, object oriented, or functional style. The material covered in Section 1 was more of a procedural style. In this section, you will learn the basics of objected oriented programming in python.

At the above reference site, read up to and including Section 9.5 (feel free to read more if you are interested). It is highly recommended that you manually write the code when following the examples.

Section 2: Function Objects

Estimated time: 1 hour

In Python, functions are first-class objects in python. This means that you can pass functions around. For example, you can a list of functions or a dictionary where the keys or values are functions. To reference the function object, simply exclude the parenthesis after the function name (see the examples below).

Being able to use functions as objects allows you to pass them as an input argument into another function. In fact, you have already done this in the first tutorial. Recall the following code:

```
strs = ['ccc', 'aaaa', 'd', 'bb']
print sorted(strs, key=len)    # ['d', 'bb', 'ccc', 'aaaa']
```

Here, the `len` function, which calculates the length of certain objects, is passed as the `key` input argument of the `sorted` function. Both the `len` function and the `sorted` function are built-in functions, and there are a variety of other built-in functions some of which accept function objects as input arguments (e.g., filter, reduce, map).

```
def square(x):
    return x**2

map(square, [1, 2, 3, 4])    # Returns [1, 4, 9, 16]
```

Above is an example using the `map` function, which applies some function (first input argument) to every item in some iterable (second input argument) and returns a list of the results. It is also easy to make custom functions that function objects as input. For example, let’s make a function that will return the “best” number from a list of positive integers, where the “best” number will be determined based on some user defined function.

```
def num_digits(x):
    return len(str(x))

def most_digits(L):
    L = sorted(L, key=num_digits)
    return L[-1]

def largest_two_digit_even(L):
    two_digit_numbers = [i for i in L if num_digits(i)==2]
    evens = [i for i in two_digit_numbers if i%2==0]
    evens.sort()
    return evens[-1]

def best(L, criteria):
    return criteria(L)

L = [1, 76, 84, 95, 214, 1023, 511, 32]
print(best(L, min))                        # Prints 1
print(best(L, largest_two_digit_even))     # Prints 84
print(best(L, most_digits))                # Prints 1023
```

The above code relies on passing several function objects around. We create two functions, `most_digits` and `largest_two_digit`, that return a single number from a list of numbers (these functions rely on the `num_digits`). These two functions are then passed to `best`, which then calls these function by simply appending parentheses, just like a normal function call. Note that the `min` function is not defined as this is a built-in function.

Now it is time for some practice. Your task is to create two new functions, `num_ones_in_binary` and `most_ones_in_binary`. (These two functions are analogous to num_digits and most_digits.) The `num_ones_in_binary` function should take a single integer (base 10) argument as input and return the number of ones in that number when converted to binary. The `most_ones_in_binary` should take a list of integers (base 10) as input and return the integer from that list that has the most ones in binary. Finally, add a final print statement that uses the `most_ones_in_binary` function on list L. Hint: there is likely a built-in function to help with the base 10 to base 2 conversion.

Submission Instructions:

- Create a `function_passing.py` file with the above code and your additions. Running python function_passing.py should result in 1, 84, 1023, and X being printed to the console, where X is result of your additions.

Section 3: Lambda Expressions

Estimated time: 0.5 hour
Reference Site: https://docs.python.org/2/tutorial/controlflow.html#lambda-expressions

Lambda expressions are a useful way to quickly pass small, “anonymous” functions wherever a function object is required. An anonymous function is a function that does not have a name, which is perfect for a throwaway-type function that will only be used once or is extremely simple. Review the small section of material from the reference site above to learn more about lambda functions.

To further solidify your understanding, consider the example code from section 2 where the square function was mapped to the list [1,2,3,4].

```
def square(x):
    return x**2

map(square, [1, 2, 3, 4])    # Returns [1, 4, 9, 16]
```

Recall that map takes a function object and iterable (e.g., list) as input arguments. Since square is such a small function that probably won't be used again (since we can just use x**2) and map requires a function object, this is a good situation for a lambda expression. As you are probably aware by now, a lambda expression uses the following syntax "lambda <argument_list> : <expression>", where expression will be returned (but you do not explicitly state "return"). Therefore, the above code could be equivalently written using a lambda expression as follows:

```
map(lambda x: x**2, [1, 2, 3, 4])    # Returns [1, 4, 9, 16]
```

It is perfectly fine to never use lambda functions and always define separate functions that you pass whenever needed. They are just another tool for adding some syntactic sugar. However, the number of times you should appropriately use lambda functions is probably quite small. For instance, although they make sense in the above example with the `map` function, realistically this whole situation should be avoided entirely by simply using list comprehension.

```
[i**2 for i in [1,2,3,4]]    # Returns [1, 4, 9, 16]
```

Moreover, the operator module has many functions that can be used in place of lambda expressions. For example, `lambda x,y: x+y` could be replaced with `operator.add`. Nevertheless, lambda functions are something that may pop up from time to time, so you should be aware of their syntax and their potential uses in your own code.

Section 4: Variable Number of Function Arguments

Estimated time: 0.5 hour
Reference Site: https://docs.python.org/2/tutorial/controlflow.html#arbitrary-argument-lists

In Python, the `*` operator is used in a function argument to specify a variable number of input arguments. Conventionally, this operator is followed by a variable named `args`.

```
def many_args(*args):
    print(args)

many_args(1, 2)        # Prints (1,2)
many_args(1, 2, 3)     # Prints (1,2,3)
```

You can also combine this feature with required arguments, by specifying the required arguments first followed by `*args`.

```
def add(a, b, *args):
    print( a + b + sum(args) )

add(1, 2)               # Prints 3
add(1, 2, 3, 4, 5)      # Prints 15
```

Also, note that the star operator can be used to split an iterable (e.g., list, set, dictionary) into its individual components. Meaning, passing two parameters into the add function above, could be done equivalently via `add(*(10,13))` or `add(10,13)`, although, in this case, there is no reason to.

In addition to the `*` operator, there is also the `**` operator. Conventionally, this operator is followed by the variable named `kwargs`. This operator allows you to pass keyword arguments into functions. Recall that keyword arguments are those arguments that are assigned some variable name in the function declaration. For example, the function sorted contains the keyword argument `reversed`, which you can specify as either True or False (e.g., `sorted(L, reversed=True)`). Using `**kwargs` allows your function to have unlimited keyword arguments that you can then reference, inside the function, by treating kwargs as a dictionary.

```
def myfunc(**kwargs):
    for k,v in kwargs.iteritems():
        print("%s -> %s" % (k, v))

myfunc(somevar="blah", anothervar=20, blah=[1,2,3])
```

Miscellaneous Links

- https://docs.python.org/2/tutorial/index.html - large python tutorial detailing how to use many things.
- http://www.pythontutor.com/ - step-by-step visualization of what a snippet of python code is doing.
- https://pypi.python.org/pypi - python package index (analogous to CRAN for R), which contains many user-created modules that can be useful.
- http://www.numpy.org/ and http://matplotlib.org/ - useful for scientific analysis and visualization.